```
$ ls
08_packets.html                    Numerals first
TOC.sh                             Uppercase next
calendar                           Then lowercase
cptodos.sh
dept.lst
emp.lst
helpdir
progs
usdsk06x
usdsk07x
usdsk08x
```

What you see here is a complete list of filenames in the current directory arranged in **ASCII collating sequence** (numbers first, uppercase and then lowercase), with one filename in each line. It includes directories also, and if you are using Linux, you would probably see the directories and ordinary files in different colors.

LINUX: If your Linux system doesn't show these colors, make sure that you create this *alias* after logging in:

```
alias ls='ls --color=tty'
```

Aliases are discussed in Section 10.4, but note that they are not supported by the Bourne shell.

Directories often contain many files, and you may simply be interested in only knowing whether a particular file is available. In that case, just use **ls** with the filename:

```
$ ls calendar
calendar
```

and if **perl** isn't available, the system clearly says so:

```
$ ls perl
perl: No such file or directory
```

ls can also be used with multiple filenames, and has options that list most of the file attributes. In the following sections, you'll see some of these options.

4.11.1 ls Options

ls has a large number of options (Table 4.1), but in this chapter, we'll present a handful of them. The other options will be taken up in later chapters. The section numbers are appropriately indicated in the table.

Output in Multiple Columns (-x) When you have several files, it's better to display the filenames in multiple columns. Modern versions of **ls** do that by default (i.e., when used without options), but if that doesn't happen on your system, use the **-x** option to produce a multicolumnar output

```
$ ls -x
```

```
08_packets.html  TOC.sh          calendar         cptodos.sh
dept.lst         emp.lst         helpdir          progs
usdsk06x         usdsk07x        usdsk08x         ux2nd06
```

If your system needs to use the -x option to display multicolumnar output, you can later customize the command to display in this format by default *(10.4)*.

Identifying Directories and Executables (-F) The output of **ls** that you have seen so far merely showed the filenames. You didn't know how many of them, if any, were directory files. To identify directories and executable files, the -F option should be used. Combining this option with -x produces a multicolumnar output as well:

```
$ ls -Fx
08_packets.html  TOC.sh*         calendar*        cptodos.sh*
dept.lst         emp.lst         helpdir/         progs/
usdsk06x         usdsk07x        usdsk08x         ux2nd06
```

Note the use of two symbols, * and /, as type indicators. The * indicates that the file contains executable code and the / refers to a directory. You can now identify the two subdirectories in the current directory—helpdir and progs.

Showing Hidden Files Also (-a) **ls** doesn't normally show all files in a directory. There are certain hidden files (filenames beginning with a dot), often found in the home directory, that normally don't show up in the listing. The -a option (all) lists all hidden files as well:

```
$ ls -axF
./               ../              .exrc            .kshrc
.profile         .rhosts          .sh_history      .xdtsupCheck
.xinitrc         08_packets.html* TOC.sh*          calendar*
  .....
```

The file .profile contains a set of instructions that are performed when a user logs in. It is conceptually similar to AUTOEXEC.BAT of DOS, and you'll know more about it later. The other file, .exrc, contains a sequence of startup instructions for the **vi** editor. To display these hidden filenames, you can either use the -a option or specify the filenames in the command line.

The first two files (. and ..) are special directories. Recall that we used the same symbols in relative pathnames to represent the current and parent directories *(4.10.1)*. These symbols have the same meaning here. Whenever you create a subdirectory, these "invisible" directories are created automatically by the kernel. You can't remove them, nor can you write into them. They help in holding the file system together.

Note: All filenames beginning with a dot are displayed only when **ls** is used with the -a option. The directory . represents the current directory and .. signifies the parent directory.

Listing Directory Contents In the last example, you specified some ordinary filenames to **ls** to have a selective listing. However, the situation will be quite different if you specify the two directory names, helpdir and progs, instead:

```
$ ls -x helpdir progs
helpdir:
forms.obd    graphics.obd    reports.obd

progs:
array.pl    cent2fah.pl    n2words.pl    name.pl
```

This time the *contents* of the directories are listed, consisting of the Oracle documentation in the `helpdir` directory and a number of **perl** program files in progs. Note that `ls`, when used with directory names as arguments, doesn't simply show their names as it does with ordinary files.

Recursive Listing (-R) The `-R` (recursive) option lists all files and subdirectories in a directory tree. Similar to the **DIR /S** command of DOS, this traversal of the directory tree is done recursively until there are no subdirectories left:

```
$ ls -xR
08_packets.html  TOC.sh          calendar        cptodos.sh
dept.lst         emp.lst         helpdir         progs
usdsk06x         usdsk07x        usdsk08x        ux2nd06

./helpdir:
forms.hlp        graphics.hlp    reports.hlp

./progs:
array.pl         cent2fah.pl     n2words.pl      name.pl
```

The list shows the filenames in three sections—the ones under the home directory and those under the subdirectories `helpdir` and progs. Note the subdirectory naming conventions followed: `./helpdir` indicates that `helpdir` is a subdirectory under `.` (the current directory). Since `/home/kumar` happens to be the current directory, the absolute pathname of this file expands to `/home/kumar/helpdir`.

Table 4.1 Options to `ls`

Option	*Description*
`-x`	Multicolumnar output
`-F`	Marks executables with *, directories with / and symbolic links with @
`-a`	Shows all filenames beginning with a dot including . and ..
`-R`	Recursive list
`-r`	Sorts filenames in reverse order (ASCII collating sequence by default)
`-l`	Long listing in ASCII collating sequence showing seven attributes of a file *(6.1)*
`-d` *dirname*	Lists only *dirname* if *dirname* is a directory *(6.2)*
`-t`	Sorts filenames by last modification time *(11.6)*
`-lt`	Sorts listing by last modification time *(11.6)*
`-u`	Sorts filenames by last access time *(11.6)*
`-lu`	Sorts by ASCII collating sequence but listing shows last access time *(11.6)*
`-lut`	As above but sorted by last access time *(11.6)*
`-i`	Displays inode number *(11.1)*

5

Handling Ordinary Files

The last chapter examined the tools that handle directories. But users actually do most of their work with ordinary (or regular) files, and it's natural that the UNIX system should feature a host of commands to handle them. Although all of these commands use filenames as arguments, they were not designed *only* to read files. In fact, many of them don't need to read a file at all. However, to understand their basic functionality, we'll use them with filenames in this chapter.

We'll first consider the common file-handling commands that the DOS environment also offers, except that the UNIX variety has more features. We'll also discuss those commands that show differences between two files and convert files between DOS and UNIX formats. Finally, we'll examine the important compression utilities with which we handle documents and software found on the Internet. As we progressively discover the shell's features in later chapters, we'll learn to use the same commands in other ways.

What You Will Learn

- View text files with **cat** and **more** (or **less**)
- Use **cat** to create a file.
- The essential file functions—copy with **cp**, remove with **rm** and rename with **mv**.
- Print a file with **lp** (**lpr** in Linux).
- Classify files with **file**.
- Count the number of lines, words and characters with **wc**.
- Display the ASCII octal value of text with **od**.
- Compare two files with **cmp**, **comm** and **diff**.
- Compress and decompress files with **gzip** and **gunzip**.
- Create an *archive* comprising multiple files with **tar**.
- Perform both functions (compressing and archiving) with **zip** and **unzip**.

Topics of Special Interest

- A discussion on the issues related to file compression and archival.
- Convert between UNIX and DOS files with **unix2dos** and **dos2unix**.
- How It Works: A graphic showing how a directory is affected by **cp**, **mv** and **rm**.

5.1 cat: DISPLAYING AND CREATING FILES

cat is one of the most well-known commands of the UNIX system. It is mainly used to display the contents of a small file on the terminal:

```
$ cat dept.lst
01|accounts|6213
02|progs|5423
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

cat, like several other UNIX commands, also accepts more than one filename as arguments:

```
cat chap01 chap02
```

The contents of the second file are shown immediately after the first file without any header information. In other words, **cat** concatenates the two files—hence its name.

5.1.1 cat Options (-v and -n)

There are two **cat** options that you may find useful, though POSIX doesn't require **cat** to support either one.

Displaying Nonprinting Characters (-v) **cat** is normally used for displaying text files only. Executables, when seen with **cat**, simply display junk. If you have nonprinting ASCII characters in your input, you can use **cat** with the **-v** option to display these characters.

Numbering Lines (-n) The **-n** option numbers lines. C compilers indicate the line number where errors are detected, and this numbering facility often helps a programmer in debugging programs. But then your **vi** editor can show line numbers too, and if your version of **cat** doesn't support **-n**, you can use the **pr** *(12.2)* command to do the same job.

5.1.2 Using cat to Create a File

cat is also useful for creating a file. Though the significance of the following sequence can be appreciated only after reading Section 8.5.2, you should now know how to create small files. Enter the command **cat**, followed by the > (the right chevron) character and the filename (for example, **foo**):

```
$ cat > foo
A > symbol following the command means that the
output goes to the filename following it. cat used
in this way represents a rudimentary editor.
[Ctrl-d]
$ _                                    Prompt returns
```

When the command line is terminated with *[Enter]*, the prompt vanishes. **cat** now waits to take input from the user. Enter the three lines, each followed by *[Enter]*. Finally press *[Ctrl-d]* to signify the end of input to the system. This is the eof character used by UNIX systems and is shown in the

stty output *(3.13)*. When this character is entered, the system understands that no further text input will be made. The file is written and the prompt returned. To verify this, simply "cat" this file:

```
$ cat foo
A > symbol following the command means that the
output goes to the filename following it. cat used
in this way represents a rudimentary editor.
```

Note: The *[Ctrl-d]* character is used to terminate input not only with **cat**, but with all commands that accept input from the keyboard.

---

cat is a versatile command. It can be used to create, display, concatenate and append to files. More importantly, it doesn't restrict itself to handling files only; it also acts on a *stream*. You can supply the input to **cat** not only by specifying a filename, but also from the output of another command. You'll learn about all this in Chapter 8.

5.2 cp: COPYING A FILE

The **cp** (copy) command copies a file or a group of files. It creates an exact image of the file on disk with a different name. The syntax requires at least two filenames to be specified in the command line. When both are ordinary files, the first is copied to the second:

```
cp chap01 unit1
```

If the destination file (`unit1`) doesn't exist, it will first be created before copying takes place. If not it will simply be overwritten without any warning from the system. So be careful when you choose your destination filename. Just check with the **ls** command whether or not the file exists.

If there is only one file to be copied, the destination can be either an ordinary or directory. You then have the option of choosing your destination filename. The following example shows two ways of copying a file to the progs directory:

```
cp chap01 progs/unit1        chap01 copied to unit1 under progs
cp chap01 progs              chap01 retains its name under progs
```

cp is often used with the shorthand notation, . (dot), to signify the current directory as the destination. For instance, to copy the file `.profile` from `/home/sharma` to your current directory, you can use either of the two commands:

```
cp /home/sharma/.profile .profile    Destination is a file
cp /home/sharma/.profile .           Destination is the current directory
```

Obviously, the second one is preferable because it requires fewer keystrokes.

cp can also be used to copy more than one file with a single invocation of the command. In that case, the last filename *must* be a directory. For instance, to copy the files `chap01`, `chap02` and `chap03` to the progs directory, you have to use **cp** like this:

```
cp chap01 chap02 chap03 progs
```

The files retain their original names in progs. If these files are already resident in progs, they will be overwritten. For the above command to work, the progs directory must exist because **cp** won't create it.

You have already seen (5.3) how the UNIX system uses the * to frame a pattern for matching more than one filename. If there were only three filenames in the current directory having the common string chap, you can compress the above sequence using the * as a suffix to chap:

```
cp chap* progs                    Copies all files beginning with chap
```

We'll continue to use the * as a shorthand for multiple filenames. There are other metacharacters too, and they are discussed in complete detail in Section 8.3.

Note: In the previous example, **cp** doesn't look for a file named chap*. Before it runs, the shell expands chap* to regenerate the command line arguments for **cp** to use.

Caution: **cp** overwrites without warning the destination file if it exists! Run **ls** before you use **cp** unless you are sure that the destination file doesn't exist or deserves to be overwritten.

5.2.1 cp Options

Interactive Copying (-i) The -i (interactive) option warns the user before overwriting the destination file. If unit1 exists, **cp** prompts for a response:

```
$ cp -i chap01 unit1
cp: overwrite unit1 (yes/no)? y
```

A y at this prompt overwrites the file, any other response leaves it uncopied.

Copying Directory Structures (-R) Many UNIX commands are capable of **recursive** behavior. This means that the command can descend a directory and examine all files in its subdirectories. The **cp -R** command behaves recurvely to copy an entire directory structure, say, progs to newprogs:

```
cp -R progs newprogs              newprogs must not exist
```

Attention! How **cp** behaves here depends on whether newprogs also exists as a directory. If newprogs doesn't exist, **cp** creates it along with the associated subdirectories. But if newprogs exists, progs becomes a subdirectory under newprogs. This means that the command run twice in succession will produce different results!

Caution: Sometimes, it's not possible to copy a file. This can happen if it's read-protected or the destination file or directory is write-protected. File permissions are discussed in Section 6.4.

5.3 rm: DELETING FILES

The **rm** (remove) command deletes one or more files. It normally operates silently and should be used with caution. The following command deletes three files:

```
rm chap01 chap02 chap03                    rm chap* could be dangerous to use!
```

A file once deleted can't be recovered. `rm` won't *normally* remove a directory, but it can remove files from one. You can remove two chapters from the `progs` directory without having to "cd" to it:

```
rm progs/chap01 progs/chap02               Or rm progs/chap0[12]
```

You may sometimes need to delete all files in a directory as part of a cleanup operation. The *, when used by itself, represents all files, and you can then use `rm` like this:

```
$ rm *                                     All files gone!
$ _
```

DOS users, beware! When you delete files in this fashion, the system won't prompt you with the message `All files in directory will be deleted!` before removing the files! The $ prompt will return silently; the work has been done. The * used here is equivalent to *.* used in DOS.

Note: Whether or not you are able to remove a file depends, not on the file's permissions, but on the permissions you have for the *directory*. Directory permissions are taken up in Section 6.6.

5.3.1 `rm` Options

Interactive Deletion (-i) Like in `cp`, the `-i` (interactive) option makes the command ask the user for confirmation before removing each file:

```
$ rm -i chap01 chap02 chap03
rm: remove chap01 (yes/no)? ?y
rm: remove chap02 (yes/no)? ?n
rm: remove chap03 (yes/no)? [Enter]        No response—file not deleted
```

A y removes the file, any other response leaves the file undeleted.

Recursive Deletion (-r or -R) With the `-r` (or `-R`) option, `rm` performs a tree walk—a thorough recursive search for all subdirectories and files within these subdirectories. At each stage, it deletes everything it finds. `rm` *won't normally remove directories, but when used with this option, it will.* Therefore, when you issue the command

```
rm -r *                                    Behaves partially like rmdir
```

you'll delete all files in the current directory and all its subdirectories. If you don't have a backup, then these files will be lost forever.

Forcing Removal (-f) `rm` prompts for removal if a file is write-protected. The `-f` option overrides this minor protection and forces removal. When you combine it with the `-r` option, it could be the most risky thing to do:

```
rm -rf *                                   Deletes everything in the current directory and below
```

Caution: Make sure you are doing the right thing before you use `rm *`. Be doubly sure before you use `rm -rf *`. The first command removes only ordinary files in the current directory. The second one removes everything—files and directories alike. If the root user (the super user) invokes `rm -rf *` in the / directory, the entire UNIX system will be wiped out from the hard disk!

5.4 mv: RENAMING FILES

The `mv` command renames (moves) files. It has two distinct functions:

- It renames a file (or directory).
- It moves a group of files to a different directory.

`mv` doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming. To rename the file `chap01` to `man01`, you should use

```
mv chap01 man01
```

If the destination file doesn't exist, it will be created. For the above example, `mv` simply replaces the filename in the existing directory entry with the new name. By default, `mv` doesn't prompt for overwriting the destination file if it exists. So be careful again.

Like `cp`, a group of files can be moved to a directory. The following command moves three files to the `progs` directory:

```
mv chap01 chap02 chap03 progs
```

`mv` can also be used to rename a directory, for instance, `pis` to `perdir`:

```
mv pis perdir
```

Like in `cp -R`, there's a difference in behavior depending on whether `perdir` exists or not. You can check that out for yourself.

There's a `-i` option available with `mv` also, and behaves exactly like in `cp`. The messages are the same and require a similar response.

HOW IT WORKS: How a Directory is Affected by cp, mv and rm

`cp`, `mv` and `rm` work by modifying the directory entries of the files they access. As shown in Fig. 5.1, `cp` adds an entry to the directory with the name of the destination file and inode number that is allotted by the kernel. `mv` replaces the name of an existing directory entry without disturbing its inode number. `rm` removes from an entry from the directory.

This is a rather simplistic view, and is true only when source and destination are in the same directory. When you "mv" a file to a directory that resides on a separate hard disk, the file is *actually* moved. You'll appreciate this better after you have understood how multiple file systems create the illusion of a single file system on your UNIX machine.

file correctly identifies the basic file types (regular, directory or device). For a regular file, it attempts to classify it further. Using the * to signify all files, this is how **file** behaves on this system having regular files of varying types:

```
$ file *
CW6.Eval.exe:    DOS executable (EXE)
User_Guide.ps:   PostScript document
archive.tar.gz: gzip compressed data - deflate method , original file name
create_user.sh: commands text
fatal.o:         ELF 32-bit MSB relocatable SPARC Version 1
fork1.c:         c program text
os_Lec03.pdf:    Adobe Portable Document Format (PDF) v1.2
```

This command identifies the file type by examining the **magic number** that is embedded in the first few bytes of the file. Every file type has a unique magic number. **file** recognizes text files, and can distinguish between shell programs, C source and object code. It also identifies DOS executables, compressed files, PDF documents and even empty files. While this method of identifying files is not wholly accurate, it's a reliable indicator.

5.8 wc: COUNTING LINES, WORDS AND CHARACTERS

UNIX features a universal word-counting program that also counts lines and characters. It takes one or more filenames as arguments and displays a four-columnar output. Before you use **wc** on the file **infile**, just use **cat** to view its contents:

```
$ cat infile
I am the wc command
I count characters, words and lines
With options I can also make a selective count
```

You can now use **wc** without options to make a "word count" of the data in the file:

```
$ wc infile
      3      20     103 infile
```

wc counts 3 lines, 20 words and 103 characters. The filename has also been shown in the fourth column. The meanings of these terms should be clear to you as they are used throughout the book:

- A **line** is any group of characters not containing a newline.
- A **word** is a group of characters not containing a space, tab or newline.
- A **character** is the smallest unit of information, and includes a space, tab and newline.

wc offers three options to make a specific count. The **-l** option counts only the number of lines, while the **-w** and **-c** options count words and characters, respectively:

```
$ wc -l infile
       3 infile                    Number of lines
$ wc -w infile
      20 infile                    Number of words
$ wc -c infile
     103 infile                    Number of characters
```

When used with multiple filenames, **wc** produces a line for each file, as well as a total count:

```
$ wc chap01 chap02 chap03
    305    4058   23179 chap01
    550    4732   28132 chap02
    377    4500   25221 chap03
   1232   13290   76532 total
```

A total as a bonus

wc, like **cat**, doesn't work with only files; it also acts on a data stream. You'll learn all about these streams in Chapter 8.

5.9 od: DISPLAYING DATA IN OCTAL

Many files (especially executables) contain nonprinting characters, and most UNIX commands don't display them properly. The file `odfile` contains some of these characters that don't show up normally:

```
$ more odfile
White space includes a
The ^G character rings a bell
The ^L character skips a page
```

The apparently incomplete first line actually contains a tab (entered by hitting *[Tab]*). The **od** command makes these commands visible by displaying the ASCII octal value of its input (here, a file). The `-b` option displays this value for each character separately. Here's a trimmed output:

```
$ od -b odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
.....Output trimmed....
```

Each line displays 16 bytes of data in octal, preceded by the offset (position) in the file of the first byte in the line. In the absence of proper mapping it's difficult to make sense of this output, but when the `-b` and `-c` (character) options are combined, the output is friendlier:

```
$ od -bc odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
          W   h   i   t   e       s   p   a   c   e       i   n   c   l
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
          u   d   e   s       a       \t  \n  T   h   e       007     c
0000040 150 141 162 141 143 164 145 162 040 162 151 156 147 163 040 141
          h   a   r   a   c   t   e   r       r   i   n   g   s       a
0000060 040 142 145 154 154 012 124 150 145 040 014 040 143 150 141 162
              b   e   l   l  \n   T   h   e      \f       c   h   a   r
  .....
```

Each line is now replaced with two. The octal representations are shown in the first line. The printable characters and escape sequences are shown in the second line. The first character in the first line is the letter `W` having the octal value 127. You'll recall having used some of the escape sequences with **echo** *(3.3)*. Let's have a look at their various representations:

- The tab character, *[Ctrl-i]*, is shown as \t and the octal value 011.
- The bell character, *[Ctrl-g]*, is shown as 007. Some systems show it as \a.
- The formfeed character, *[Ctrl-l]*, is shown as \f and 014.
- The LF (linefeed or newline) character, *[Ctrl-j]*, is shown as \n and 012. Note that **od** makes the newline character visible too.

Like **wc**, **od** also takes a command's output as its own input, and in Section 5.13, we'll use it to display nonprintable characters in filenames.

5.10 cmp: COMPARING TWO FILES

You may often need to know whether two files are identical so one of them can be deleted. There are three commands in the UNIX system that can tell you that. In this section, we'll have a look at the **cmp** (compare) command. Obviously, it needs two filenames as arguments:

```
$ cmp chap01 chap02
chap01 chap02 differ: char 9, line 1
```

The two files are compared byte by byte, and the location of the first mismatch (in the ninth character of the first line) is echoed to the screen. By default, **cmp** doesn't bother about possible subsequent mismatches but displays a detailed list when used with the **-l** (list) option.

If two files are identical, **cmp** displays no message, but simply returns the prompt. You can try it out with two copies of the same file:

```
$ cmp chap01 chap01
$ _
```

This follows the UNIX tradition of quiet behavior. This behavior is also very important because the comparison has returned a *true* value, which can be subsequently used in a shell script to control the flow of a program.

5.11 comm: WHAT IS COMMON?

Suppose you have two lists of people and you are asked to find out the names available in one and not in the other, or even those common to both. **comm** is the command you need for this work. It requires two *sorted* files, and lists the differing entries in different columns. Let's try it on these two files:

```
$ cat file1
c.k. shukla
chanchal singhvi
s.n. dasgupta
sumit chakrobarty
```

```
$ cat file2
anil aggarwal
barun sengupta
c.k. shukla
lalit chowdury
s.n. dasgupta
```

Both files are sorted and have some differences. When you run **comm**, it displays a three-columnar output:

```
$ comm file[12]                          Comparing file1 and file2
        anil aggarwal
        barun sengupta
                c.k. shukla
chanchal singhvi
        lalit chowdury
                s.n. dasgupta
sumit chakrobarty
```

The first column contains two lines unique to the first file, and the second column shows three lines unique to the second file. The third column displays two lines common (hence its name) to both files.

This output provides a good summary to look at, but is not of much use to other commands that take **comm**'s output as their input. These commands require single-column output from **comm**, and **comm** can produce it using the options -1, -2 or -3. To drop a particular column, simply use the column number as an option prefix. You can also combine options and display only those lines that are common:

```
comm -3 foo1 foo2          Selects lines not common to both files
comm -13 foo1 foo2         Selects lines present only in second file
```

The last example and one more with the other matching option (-23) has more practical value than you may think, but we'll not discuss their application in this text.

5.12 diff: CONVERTING ONE FILE TO OTHER

diff is the third command that can be used to display file differences. Unlike its fellow members **cmp** and **comm**, it also tells you which lines in one file have to be *changed* to make the two files identical. When used with the same files, it produces a detailed output:

```
$ diff file1 file2          Or diff file[12]
0a1,2                       Append after line 0 of first file
> anil aggarwal             this line
> barun sengupta            and this line
2c4                         Change line 2 of first file
< chanchal singhvi          Replacing this line
---                         with
> lalit chowdury            this line
4d5                         Delete line 4 of first file
< sumit chakrobarty         containing this line
```

diff uses certain special symbols and **instructions** to indicate the changes that are required to make two files identical. You should understand these instructions as they are used by the **ed** command, one of the most powerful commands on the system.

Each instruction uses an **address** combined with an **action** that is applied to the first file. The instruction **0a1,2** means appending two lines after line 0, which become lines 1 and 2 in the second file. **2c4** changes line 2 which is line 4 in the second file. **4d5** deletes line 4.

6

Basic File Attributes

In the previous two chapters, you created files and directories, navigated the file system, and copied, moved and removed files without any problem. In real life, however, matters may not be so rosy. You may have problems when handling a file or directory. Your file may be modified or even deleted by others. A restoration from a backup may be unable to write to your directory. You must know why these problems happen and how to rectify and prevent them.

The UNIX file system lets users access other files not belonging to them and without infringing on security. A file also has a number of *attributes* (properties) that are stored in the inode. In this chapter, we'll use the **ls -l** command with additional options to display these attributes. We'll mainly consider the two basic attributes—permissions and ownership—both of which are changeable by well-defined rules. The remaining attributes are taken up in Chapter 11.

What You Will Learn

- Interpret the significance of the seven fields of the **ls -l** output (*listing*).
- How to obtain the listing of a specific directory.
- The importance of *ownership* and *group ownership* of a file and how they affect security.
- The significance of the nine permissions of a file as applied to different *categories* of users.
- Use **chmod** to change all file permissions in a relative and absolute manner.
- Use **chown** and **chgrp** to change the owner and group owner of files on BSD and AT&T systems.

Topics of Special Interest

- The importance of directory permissions (discussion on individual permissions deferred to Chapter 11).
- An introductory treatment of the relationship that exists between file ownership, file permissions and directory permissions.

6.1 ls -l: LISTING FILE ATTRIBUTES

We have already used the **ls** command with a number of options. It's the **-l** (long) option that reveals most. This option displays most attributes of a file—like its permissions, size and ownership

details. The output in UNIX lingo is often referred to as the **listing**. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence.

`ls` looks up the file's inode to fetch its attributes. Let's use `ls -l` to list seven attributes of all files in the current directory:

```
$ ls -l
total 72
-rw-r--r--   1 kumar    metal       19514 May 10 13:45 chap01
-rw-r--r--   1 kumar    metal        4174 May 10 15:01 chap02
-rw-rw-rw-   1 kumar    metal          84 Feb 12 12:30 dept.lst
-rw-r--r--   1 kumar    metal        9156 Mar 12  1999 genie.sh
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

The list is preceded by the words `total 72`, which indicates that a total of 72 blocks are occupied by these files on disk, each block consisting of 512 bytes (1024 in Linux). We'll now briefly describe the significance of each field in the output.

File Type and Permissions The first column shows the type and *permissions* associated with each file. The first character in this column is mostly a `-`, which indicates that the file is an ordinary one. This is, however, not so for the directories `helpdir` and `progs`, where you see a `d` at the same position.

You then see a series of characters that can take the values `r`, `w`, `x` and `-`. In the UNIX system, a file can have three types of permissions—read, write and execute. You'll see how to interpret these permissions and also how to change them in Sections 6.4 and 6.5.

Links The second column indicates the number of *links* associated with the file. This is actually the number of filenames maintained by the system of that file. UNIX lets a file have as many names as you want it to have, even though there is a single file on disk. This attribute will be discussed in Chapter 11.

Note: A link count greater than one indicates that the file has more than one name. That doesn't mean that there are two copies of the file.

Ownership When you create a file, you automatically become its *owner*. The third column shows kumar as the owner of all of these files. The owner has full authority to tamper with a file's contents and permissions—a privilege not available with others except the root user. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group Ownership When opening a user account, the system administrator also assigns the user to some group. The fourth column represents the *group owner* of the file. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file. It's generally desirable that the group have a set of privileges distinct from others as well as the owner. Ownership and group ownership are elaborated in Section 6.7.

File Size The fifth column shows the size of the file in bytes, i.e., the amount of data it contains. The important thing to remember is that it is only a character count of the file and not a measure of disk space that it occupies. The space occupied by a file on disk is usually larger than this figure since files are written to disk in blocks of 1024 bytes or more. In other words, even though the file `dept.lst` contains 84 bytes, it would occupy 1024 bytes on disk on systems that use a block size of 1024 bytes. We'll discuss the significance of block size much later in the text.

The two directories show smaller file sizes (512 bytes each). This is to be expected as a directory maintains a list of filenames along with an identification number (the inode number) for each file. The size of the directory file depends on the size of this list—whatever be the size of the files themselves.

Last Modification Time The sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its contents have changed in any way. If you change only the permissions or ownership of the file, the modification time remains unchanged. If the file is less than a year old since its last modification date, the year won't be displayed. Note that the file `genie.sh` has been modified more than a year ago.

You'll often need to run automated tools that make decisions based on a file's modification time. This column shows two other time stamps when `ls` is used with other options. The time stamps are discussed in Chapter 11.

Filename The last column displays the filenames arranged in ASCII collating sequence. You already know *(4.2)* that UNIX filenames can be very long (up to 255 characters). If you would like to see an important file at the top of the listing, then choose its name in uppercase—at least, its first letter.

The order of the list can be changed by combining the `-l` option with other options. In the rest of the chapter, we'll discuss permissions and ownership, and also learn how to change them.

6.2 THE -d OPTION: LISTING DIRECTORY ATTRIBUTES

You'll recall *(4.11.1)* that `ls`, when used with directory names, lists files in the directory rather than the directory itself. To force `ls` to list the attributes of a directory, rather than its contents, you need to use the `-d` (directory) option:

```
$ ls -ld helpdir progs
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

Directories are easily identified in the listing by the first character of the first column, which here shows a `d`. For ordinary files, this slot always shows a `-` (hyphen), and for device files, either a `b` or `c`. The significance of the attributes of a directory differ a good deal from an ordinary file. Directories will be considered in some detail in Chapter 11.

Tip: To see the attributes of a directory rather than the files contained in it, use `ls -ld` with the directory name. Note that simply using `ls -d` will *not* list all subdirectories in the current directory. Strange though it may seem, `ls` has no option to list only directories!

6.3 FILE OWNERSHIP

Before we take up file permissions, you need to understand the significance of file ownership. When you create a file, your username shows up in the third column of the file's listing; you are the **owner** of the file. Your group name is seen in the fourth column; your group is the **group owner** of the file. If you copy someone else's file, you are the owner of the copy. If you can't create files in other users' home directories, it's because those directories are not owned by you (and the owner has not allowed you write access).

Several users may belong to a single group. People working on a project are generally assigned a common group, and all files created by group members (who have separate user-ids) will have the same group owner. However, make no mistake: The privileges of the group are set by the owner of the file and not by the group members.

When the system administrator creates a user account, he has to assign these parameters to the user:

- The user-id (UID)—both its name and numeric representation.
- The group-id (GID)—both its name and numeric representation.

The file `/etc/passwd` maintains the UID (both the number and name) and GID (but only the number). `/etc/group` contains the GID (both number and name). We'll discuss these two files when learning to add a user account in Chapter 15.

Tip: To know your own UID and GID without viewing `/etc/passwd` and `/etc/group`, use the `id` command:

```
$ id
uid=655537(kumar)  gid=655535(metal)
```

The uid is kumar, and if this name matches the one shown in the ownership field of the listing (`ls -l` output), then you are the owner of those files. It also means that you have the authority to change the file's attributes.

6.4 FILE PERMISSIONS

UNIX has a simple and well-defined system of assigning permissions to files. Let's issue the `ls -l` command once again to view the permissions of a few files:

```
$ ls -l chap02 dept.lst dateval.sh
-rwxr-xr--   1 kumar    metal       20500 May 10 19:21 chap02
-rwxr-xr-x   1 kumar    metal         890 Jan 29 23:17 dateval.sh
-rw-rw-rw-   1 kumar    metal          84 Feb 12 12:30 dept.lst
```

Observe the first column that represents file permissions. These permissions are also different for the three files. UNIX follows a three-tiered file protection system that determines a file's access rights. To understand how this system works, let's break up the permissions string of the file chapter into three groups. The initial - (in the first column) represents an ordinary file and is left out of the permissions string:

```
r w x     r - x     r - -
```

Each group here represents a **category** and contains three slots, representing the read, write and execute permissions of the file—in that order. `r` indicates read permission, which means **cat** can display the file. `w` indicates write permission; you can edit such a file with an editor. `x` indicates execute permission; the file can be executed as a program. The - shows the absence of the corresponding permission.

The first group (`rwx`) has all three permissions. The file is readable, writable and executable by the *owner* of the file, kumar. But do we know who the owner is? Yes we do. The third column shows kumar as the owner and the first permissions group applies to kumar. You have to log in with the username kumar for these privileges to apply to you.

The second group (`r-x`) has a hyphen in the middle slot, which indicates the absence of write permission by the *group owner* of the file. This group owner is metal, and all users belonging to the metal group have read and execute permissions only.

The third group (`r--`) has the write and execute bits absent. This set of permissions is applicable to *others*, i.e., those who are neither the owner kumar nor belong to the metal group. This category (others) is often referred to as the *world*. This file is not world-writable.

You can set different permissions for the three categories of users—owner, group and others. It is important that you understand them because a little learning here can be a dangerous thing. A faulty file permission is a sure recipe for disaster.

Note: The group permissions here don't apply to kumar (the owner) even if kumar belongs to the metal group. The owner has its own set of permissions that override the group owner's permissions. However, when kumar renounces the ownership of the file, the group permissions then apply to him.

6.5 chmod: CHANGING FILE PERMISSIONS

Before we take up **chmod**, let's decide to change a habit. Henceforth, we'll refer to the owner as *user* because that's how the **chmod** command (which changes file permissions) refers to the owner. In this section, whenever we use the term *user*, we'll actually mean *owner*.

A file or directory is created with a default set of permissions, and this default is determined by a simple setting (called *umask*), which we'll discuss later. Generally, the default setting write-protects a file from all except the user (new name for owner), though all users *may* have read access. However, this may not be so on your system. To know your system's default, create a file `xstart`:

```
$ cat /usr/bin/startx > xstart                    Actually copies the file startx
$ ls -l xstart
-rw-r--r--   1 kumar    metal        1906 Sep  5 23:38 xstart
```

It seems that, by default, a file doesn't also have execute permission. So how does one execute such a file? To do that, the permissions of the file need to be changed. This is done with **chmod**.

The **chmod** (change mode) command is used to set the permissions of one or more files for all three categories of users (user, group and others). It can be run only by the user (the owner) and the superuser. The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions.
- In an absolute manner by specifying the final permissions.

We'll consider both ways of using **chmod** in the following sections.

6.5.1 Relative Permissions

When changing permissions in a relative manner, **chmod** only changes the permissions specified in the command line *and leaves the other permissions unchanged.* In this mode it uses the following syntax:

```
chmod category operation permission filename(s)
```

chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

- User *category* (user, group, others)
- The *operation* to be performed (assign or remove a permission)
- The type of *permission* (read, write, execute)

By using suitable abbreviations for each of these components, you can frame a compact expression and then use it as an argument to **chmod**. The abbreviations used for these three components are shown in Table 6.1.

Now let's consider an example. To assign execute permission to the user (We won't remind again that user here is the owner.) of the file xstart, we need to frame a suitable expression by using appropriate characters from each of the three columns of Table 6.1. Since the file needs to be executable only by the user, the expression required is u+x:

```
$ chmod u+x xstart
$ ls -l xstart
-rwxr--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. You can now execute the file if you are the owner of the file but the other categories (i.e., group and others) still can't. To enable all of them to execute this file, you have to use multiple characters to represent the user category (ugo):

```
$ chmod ugo+x xstart ; ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
```

The string ugo combines all the three categories—user, group and others. UNIX also offers a shorthand symbol a (all) to act as a synonym for the string. And, as if that wasn't enough, there's an even shorter form that combines these three categories. When it is not specified, the permission applies to all categories. So the previous sequence can be replaced by either of the following:

```
chmod a+x xstart          a implies ugo
chmod +x xstart           By default, a is implied
```

chmod accepts multiple filenames in the command line. When you need to assign the same set of permissions to a group of files, this is what you should do:

```
chmod u+x note note1 note3
```

Permissions are removed with the - operator. To remove the read permission from both group and others, use the expression go-r:

```
$ ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
$ chmod go-r xstart ; ls -l xstart
-rwx--x--x   1 kumar    metal        1906 May 10 20:30 xstart
```

chmod also accepts multiple expressions delimited by commas. For instance, to restore the original permissions to the file xstart, you have to remove the execute permission from all (a-x) and assign read permission to group and others (go+r):

```
$ chmod a-x,go+r xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

More than one permission can also be set; **u+rwx** is a valid **chmod** expression. So setting write and execute permissions for others is no problem:

```
$ chmod o+wx xstart ; ls -l xstart
-rw-r--rwx   1 kumar    metal        1906 May 10 20:30 xstart
```

We described relative permissions here, but **chmod** also uses an absolute assignment system, which is taken up in the next topic.

Table 6.1 Abbreviations Used by **chmod**

Category	*Operation*	*Permission*
u—User	+—Assigns permission	r—Read permission
g—Group	-—Removes permission	w—Write permission
o—Others	=—Assigns absolute permission	x—Execute permission
a—All (ugo)		

6.5.2 Absolute Permissions

Sometimes you don't need to know what a file's current permissions are, but want to set all nine permission bits explicitly. The expression used by **chmod** here is a string of three octal numbers (base 8). You may or may not be familiar with this numbering system, so a discussion would be in order.

Octal numbers use the base 8, and octal digits have the values 0 to 7. This means that a set of three bits can represent one octal digit. If we represent the permissions of each category by one octal digit, this is how the permissions can be represented:

- Read permission—4 (Octal 100)
- Write permission—2 (Octal 010)
- Execute permission—1 (Octal 001)

For each category we add up the numbers. For instance, 6 represents read and write permissions, and 7 represents all permissions as can easily be understood from Table 6.2.

Table 6.2

Binary	*Octal*	*Permissions*	*Significance*
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely. The most significant digit represents user and the least one represents others. **chmod** can use this three-digit string as the expression.

You can use this method to assign read and write permission to all three categories. Without octal numbers, you should normally be using **chmod a+rw xstart** to achieve the task (assuming there was no execute permission initially). Now you can use a different method:

```
$ chmod 666 xstart ; ls -l xstart
-rw-rw-rw-   1 kumar    metal        1906 May 10 20:30 xstart
```

The 6 indicates read and write permissions (4 + 2). To restore the original permissions to the file, you need to remove the write permission (2) from group and others:

```
$ chmod 644 xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

Caution: If a directory has write permission for group and others also, be assured that every user can remove every file in the directory! As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Note: The default file and directory permissions on your machine could be different from what has been assumed here. The defaults are determined by the *umask* setting of your shell. This topic is discussed in Section 11.5.

6.7 CHANGING FILE OWNERSHIP

File ownership is a feature often ignored by many users. By now you know well enough that when a user kumar of the metal group creates a file foo, he becomes the owner of foo, and metal becomes the group owner. It's only kumar who can change the major file attributes like its permissions and group ownership. No member of the metal group (except kumar) can change these attributes. However, when sharma copies foo, the ownership of the copy is transferred to sharma, and he can then manipulate the attributes of the copy at will.

There are two commands meant to change the ownership of a file or directory—**chown** and **chgrp**. UNIX systems differ in the way they restrict the usage of these two commands. On BSD-based systems, only the system administrator can change a file's owner with **chown**. On the same systems the restrictions are less severe when it comes to changing groups with **chgrp**. On other systems only the owner can change both.

6.7.1 chown: Changing File Owner

We'll first consider the behavior of BSD-based **chown** (change owner) that has been adopted by many systems including Solaris and Linux. The command is used in this way:

```
chown options owner [:group] file(s)
```

chown transfers ownership of a file to a user, and it seems that it can optionally change the group as well. The command requires the user-id (UID) of the recipient, followed by one or more filenames. Changing ownership requires superuser permission, so let's first change our status to that of superuser with the **su** command:

```
$ su
Password: ********          This is the root password!
# _                         This is another shell
```

After the password is successfully entered, **su** returns a # prompt, the same prompt used by root. **su** lets us acquire superuser status if we know the root password. To now renounce the ownership of the file note to sharma, use **chown** in the following way:

```
# ls -l note
-rwxr----x   1 kumar    metal        347 May 10 20:30 note
# chown sharma note ; ls -l note
-rwxr----x   1 sharma   metal        347 May 10 20:30 note
```

```
# exit                                  Switches from superuser's shell
$ _                                     to user's login shell
```

Once ownership of the file has been given away to sharma, the user file permissions that previously applied to kumar now apply to sharma. Thus, kumar can no longer edit note since there's no write privilege for group and others. He can't get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

6.7.2 chgrp: Changing Group Owner

By default, the group owner of a file is the group to which the owner belongs. The **chgrp** (change group) command changes a file's group owner. On systems that implement the BSD version of **chgrp** (like Solaris and Linux), a user can change the group owner of a file, but only to a group to which she also belongs. Yes, a user can belong to more than one group, and the one shown in /etc/passwd is the user's main group. We'll discuss *supplementary groups* in Chapter 15 featuring system administration.

chgrp shares a similar syntax with **chown**. In the following example, kumar changes the group ownership of dept.lst to dba (no superuser permission required):

```
$ ls -l dept.lst
-rw-r--r--   1 kumar    metal          139 Jun  8 16:43 dept.lst
$ chgrp dba dept.lst ; ls -l dept.lst
-rw-r--r--   1 kumar    dba            139 Jun  8 16:43 dept.lst
```

This command will work on a BSD-based system if kumar is also a member of the dba group. If he is not, then only the superuser can make the command work. Note that kumar can reverse this action and restore the previous group ownership (to metal) because he is still owner of the file and consequently retains all rights related to it.

Using **chown** *to Do Both* As an added benefit, UNIX allows the administrator to use only **chown** to change both owner and group. The syntax requires the two arguments to be separated by a :

```
chown sharma:dba dept.lst               Ownership to sharma, group to dba
```

Like **chmod**, both **chown** and **chgrp** use the **-R** option to perform their operations in a recursive manner.

6.8 CONCLUSION

In this chapter we considered two important file attributes—permissions and ownership. After we complete the first round of discussions related to files, we'll take up the other file attributes—the links and time stamps (Chapter 15). We also need to examine the inode that stores all of these attributes. In the next chapter, we'll discuss file editing with the **vi** editor.

WRAP UP

file attributes, and the output is often called the *listing*. ls -ld directory attributes.